MANUAL TÉCNICO

Allan Ricardo Barillas Sosa – 201906572
Facultad de Ingeniería - Ciencias y Sistemas
Universidad de San Carlos de Guatemala

Sistemas Organizacionales y Gerenciales 2

Sección P

05 de mayo de 2025

Objetivo General:

Desarrollar un modelo de aprendizaje automático capaz de reconocer gestos faciales y físicos (como señas con las manos) mediante imágenes procesadas desde cámara en tiempo real.

Objetivos Específicos:

- Capturar imágenes clasificadas por tipo de gesto utilizando OpenCV.
- Crear un dataset organizado y balanceado para el entrenamiento supervisado.
- Entrenar un modelo con TensorFlow para la clasificación de gestos.
- Evaluar la precisión del modelo y realizar predicciones en tiempo real.

Entrenamiento de modelos

Descripción de los componentes y métodos utilizados

- **Lenguaje principal:** Python 3.10
- **Bibliotecas utilizadas:**
  - OpenCV: Para captura de imágenes desde cámara.
  - TensorFlow/Keras: Para construir, entrenar y evaluar el modelo.
  - NumPy: Para manejo de arreglos y estructuras de datos.
  - Matplotlib: Para visualización de métricas y resultados.
- **Estructura general del sistema:**
  - Captura y almacenamiento de imágenes en carpetas por clase.
  - Preprocesamiento de imágenes (redimensionamiento, normalización).
  - Entrenamiento de modelo CNN con Keras.
  - Evaluación y predicción de gestos.

MODELO DE GESTOS

Pasos utilizados para la creación del dataset

Paso 1: Definición de clases de gestos
Se definieron clases como:
feliz, enojado, neutral, asco, miedo, ttristeza.

Paso 2: Captura de imágenes
Se creó un script Python para capturar imágenes desde la webcam:

```
import cv2
import os
import datetime

# Lista de clases de gestos
gestos = ["a_feliz", "v_sorpresa", "c_neutral", "d_enojado", "e_asco", "f_miedo", "g_tristeza"]

# Crear carpetas para cada gesto si no existen
for gesto in gestos:
    os.makedirs(f'gestos2/{gesto}', exist_ok=True)

# Captura de video desde la cámara (NO forzamos resolución)
cap = cv2.VideoCapture(1, cv2.CAP_DSHOW)
```

```
    gesto_actual = gestos[0]
    print(f"Capturando imágenes para: {gesto_actual}")
    print("Presiona G para cambiar de gesto, S para guardar imagen, Q para salir.")

    indice_gesto = 0
    contador = len(os.listdir(f'gestos2/{gesto_actual}'))

    while True:
        ret, frame = cap.read()
        if not ret:
            print("Error al acceder a la cámara.")
            break



        cv2.imshow("Captura de Gestos", frame)

        key = cv2.waitKey(1) & 0xFF

        if key == ord('q'):  # Salir
            break
        elif key == ord('s'):  # Guardar imagen con resolución original
            timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S%f")
            ruta = f"gestos2/{gesto_actual}/{gesto_actual}_{timestamp}.png"
            cv2.imwrite(ruta, frame)
            print(f"Imagen guardada en {ruta}")
            contador += 1
            cv2.waitKey(300)  # Evitar doble captura
        elif key == ord('g'):  # Cambiar gesto
            indice_gesto = (indice_gesto + 1) % len(gestos)
            gesto_actual = gestos[indice_gesto]
            contador = len(os.listdir(f'gestos2/{gesto_actual}'))
            print(f"Gesto cambiado a: {gesto_actual}")

    cap.release()
    cv2.destroyAllWindows()
```

Estructura del dataset

Gestos2/
├— feliz/
├— enojado/
├— ....

Comandos utilizados para entrenar el modelo

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import numpy as np
import os
from sklearn.model_selection import train_test_split

ruta_base = "gestos2"  # tu nueva carpeta
tamanio_img = (400, 400)  # mejor resolución para precisión
clases = sorted(os.listdir(ruta_base))  # Detecta clases automáticamente

imagenes = []
etiquetas = []

for i, clase in enumerate(clases):
    carpeta = os.path.join(ruta_base, clase)
    for archivo in os.listdir(carpeta):
        if archivo.lower().endswith((".png", ".jpg", ".jpeg")):
            img_path = os.path.join(carpeta, archivo)
            img = load_img(img_path, color_mode='grayscale', target_size=tamanio_img)  # <--- aún se
redimensiona
            img_array = img_to_array(img) / 255.0
            imagenes.append(img_array)
            etiquetas.append(i)


# Convertir a arrays
X = np.array(imagenes).reshape(-1, tamanio_img[0], tamanio_img[1], 1)
y = np.array(etiquetas)

# Dividir
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Definir modelo
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=(tamanio_img[0], tamanio_img[1], 1)),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D(2, 2),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(len(clases), activation='softmax')  # Salida según número de clases
])
```

```
model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))

# Guardar el modelo
model.save("modelo_gestos_custom.keras")
model.save("modelo_gestos_custom.h5")
```

Conversion de h5 a tensorflowJS

(venv) PS C:\Users\Sosa\Documents\GitHub\Geren2_Proyecto\ia> tensorflowjs_converter --input_format=keras modelo_gestos_custom.h5 tfjs_model_gestos2

MODELO DE NUMEROS

datos/
├── 0/
├── 1/
├── 2/
├── 3/
├── 4/
├── 5/
├── 6/
├── 7/
├── 8/
├── 9/

Herramienta de recolección de datos con dibujo manual

Para recolectar datos manualmente, se implementó una interfaz gráfica utilizando Tkinter y PIL (Pillow). Esta herramienta permite dibujar un número del 0 al 9 con el mouse, seleccionarlo con botones tipo radio, y guardarlo automáticamente en la carpeta correspondiente. Las imágenes se redimensionan a 28x28 píxeles en escala de grises, formato ideal para modelos como CNN.

```python
import tkinter as tk
from PIL import Image, ImageDraw
import os
import datetime

# Crear carpetas para cada número si no existen
for i in range(10):
    os.makedirs(f'datos/{i}', exist_ok=True)

# Interfaz
class DibujoNumerosApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Dibujador de Números")
        self.numero_actual = tk.StringVar(value="0")

        # Canvas de dibujo
        self.canvas = tk.Canvas(root, width=280, height=280, bg='black')
        self.canvas.grid(row=0, column=0, columnspan=10)
        self.canvas.bind('<B1-Motion>', self.dibujar)

        # Imagen PIL para guardar
        self.imagen = Image.new("L", (280, 280), 'black')
        self.draw = ImageDraw.Draw(self.imagen)

        # Botones del 0 al 9
        for i in range(10):
            btn = tk.Radiobutton(root, text=str(i), variable=self.numero_actual, value=str(i))
            btn.grid(row=1, column=i)

        # Botón para guardar
        guardar_btn = tk.Button(root, text="Guardar imagen", command=self.guardar)
        guardar_btn.grid(row=2, column=3, columnspan=2)

        # Botón para limpiar
        limpiar_btn = tk.Button(root, text="Limpiar", command=self.limpiar)
        limpiar_btn.grid(row=2, column=5, columnspan=2)

    def dibujar(self, event):
        x, y = event.x, event.y
        r = 8  # radio del pincel
        self.canvas.create_oval(x - r, y - r, x + r, y + r, fill='white', outline='white')
        self.draw.ellipse([x - r, y - r, x + r, y + r], fill='white')

    def guardar(self):
```

```
    numero = self.numero_actual.get()
    timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S%f")
    path = f"datos/{numero}/{numero}_{timestamp}.png"
    img_28x28 = self.imagen.resize((28, 28))
    img_28x28.save(path)
    print(f"Guardada imagen en: {path}")
    self.limpiar()

  def limpiar(self):
    self.canvas.delete('all')
    self.imagen = Image.new("L", (280, 280), 'black')
    self.draw = ImageDraw.Draw(self.imagen)

# Ejecutar
root = tk.Tk()
app = DibujoNumerosApp(root)
root.mainloop()
```

Entrenamiento del Modelo

Una vez recopiladas y organizadas las imágenes por clase (0–9), se procedió a cargar los datos, preprocesarlos y entrenar una red neuronal convolucional (CNN) utilizando TensorFlow y Keras.

Carga y preparación de datos

Se recorren las carpetas datos/0 a datos/9, leyendo cada imagen en escala de grises y redimensionándola a 28x28 píxeles. Se normalizan los valores a un rango de 0 a 1.

Se usó train_test_split de scikit-learn para separar los datos en un 80% para entrenamiento y 20% para prueba:

El modelo está basado en una **CNN simple**, con dos capas convolucionales seguidas de max pooling, aplanamiento y dos capas densas (la última con softmax para clasificación multiclase):

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import numpy as np
import os
from sklearn.model_selection import train_test_split
```

```
ruta_base = "datos"
tamanio_img = (28, 28)

imagenes = []
etiquetas = []

for etiqueta in range(10):
  carpeta = os.path.join(ruta_base, str(etiqueta))
  for archivo in os.listdir(carpeta):
    if archivo.endswith(".png"):
      img_path = os.path.join(carpeta, archivo)
      img = load_img(img_path, color_mode='grayscale', target_size=tamanio_img)
      img_array = img_to_array(img) / 255.0
      imagenes.append(img_array)
      etiquetas.append(etiqueta)

X = np.array(imagenes).reshape(-1, 28, 28, 1)
y = np.array(etiquetas)

# Dividir en entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Definir y entrenar modelo
model = tf.keras.Sequential([
  tf.keras.layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
  tf.keras.layers.MaxPooling2D(2,2),
  tf.keras.layers.Conv2D(64, (3,3), activation='relu'),
  tf.keras.layers.MaxPooling2D(2,2),
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy'])

model.fit(X_train, y_train, epochs=10, validation_data=(X_test, y_test))

# Guardar el modelo
model.save("modelo_numeros_custom.keras")
model.save("modelo_numeros_custom.h5")
#model.export("modelo_numeros_customizer")
```

Conversion de h5 a tensorflowJS

(venv) PS C:\Users\Sosa\Documents\GitHub\Geren2_Proyecto\ia> tensorflowjs_converter --input_format=keras modelo_numeros_custom.h5 tfjs_model

---

Server reconocimiento de imágenes

Para permitir la interacción del usuario con el modelo de reconocimiento de números mediante una interfaz web, se implementó un **servidor utilizando Node.js con Express y EJS**. Esta estructura facilita la carga de imágenes, la visualización del resultado y la integración con el modelo previamente entrenado, **sin necesidad de una API REST**, lo cual cumple con los lineamientos del proyecto.

Estructura del servidor

- Se utilizó **Express.js** como framework de servidor ligero.
- **EJS** se usó como motor de plantillas para generar páginas dinámicas, permitiendo renderizar directamente los resultados sin recargar toda la aplicación ni consumir endpoints REST.
- Se creó una carpeta llamada **public/** para alojar todos los archivos estáticos (scripts JavaScript, CSS, imágenes, modelos TensorFlow, etc.).

Ventajas de esta implementación

- **Evita el uso de API REST**, alineándose a los requisitos del proyecto.
- Permite una **carga sencilla del modelo .h5 o .keras en el navegador**, ya que los archivos están disponibles directamente desde el cliente mediante rutas públicas.
- Se manejan variables de entorno, rutas dinámicas y respuestas del modelo directamente desde el **servidor Node + vistas EJS**, lo cual simplifica la arquitectura.
- El usuario puede **cargar imágenes desde el navegador** y obtener la predicción de forma inmediata, con una interfaz intuitiva.

Flujo básico del sistema

1. El usuario accede a una página renderizada por EJS (index.ejs) desde el navegador.

2. La página contiene un formulario o canvas para cargar o dibujar una imagen.
3. Al enviar la imagen, esta se procesa en el navegador con JavaScript (o se envía al backend si se requiere predicción con Python vía child_process o TensorFlow.js).
4. Se carga el modelo desde la carpeta public/ y se realiza la predicción.
5. El resultado es renderizado en pantalla, sin necesidad de redirección o llamadas asíncronas a endpoints externos.

```
mi_proyecto/
├── public/
│   ├── models/
│   │   ├── modelo_numeros_custom.json
│   │   └── modelo_gestos_custom.json
│   └── css/
│       └── estilos.css
├── views/
│   └── index.ejs
├── app.js
├── package.json
```

Esta implementación permite trabajar completamente desde el cliente y el servidor sin depender de una arquitectura REST, asegurando simplicidad, rendimiento y cumplimiento de los requisitos establecidos.

Interfaz Web para Análisis de Gestos

Para facilitar la interacción directa del usuario con el sistema de reconocimiento de gestos, se diseñó una página web dinámica (analizar-gestos.ejs) renderizada por el servidor Express usando EJS. Esta vista permite capturar gestos faciales o físicos desde la cámara, o cargar imágenes desde el dispositivo, y luego procesarlas directamente en el navegador utilizando TensorFlow.js, sin necesidad de hacer peticiones a un backend REST.

Estructura general de la interfaz

- Diseño visual: El documento HTML utiliza un diseño limpio y centrado, con una cabecera (header), cuerpo principal (main) y pie de página (footer), todos con estilos personalizados en línea y mediante CSS externo (/css/estilos.css).
- Navegación: Incluye botones para navegar hacia el dashboard y la sección de análisis de números.

- Video en vivo: Se accede a la cámara del dispositivo en tiempo real mediante getUserMedia, mostrando la transmisión dentro de una etiqueta <video>.
- Captura de imagen: Al presionar el botón "Capturar Foto", se toma un cuadro de la transmisión y se renderiza en un <canvas> oculto y luego en un <img> como vista previa.
- Carga de archivo: Alternativamente, el usuario puede seleccionar una imagen desde su dispositivo usando un <input type="file">.

Procesamiento de imagen con TensorFlow.js

Una vez cargada o capturada una imagen, el usuario puede hacer clic en "Comprobar", lo que activa una función de inferencia directa desde el navegador:

```
async function comprobar() {
    const imgElement = document.getElementById('preview');

    if (!imgElement.src || imgElement.src.startsWith('data:,'))
      return alert("Captura o sube una imagen antes de comprobar.");

    const tensor = tf.tidy(() => {
      // Cargar desde el <img> (color RGB), redimensionar a 28x28
      const img = tf.browser.fromPixels(imgElement)
        .resizeNearestNeighbor([400, 400])
        .mean(2) // convierte RGB a escala de grises
        .toFloat()
        .expandDims(2) // para dejarla como [400,400,1]
        .div(255.0)    // normalización
        .expandDims(0); // para dejarla como [1,28,28,1]
      return img;
    });

    const pred = modelo.predict(tensor);
    console.log(pred);
    const data = await pred.data();
    console.log(data);

    const resultados = Array.from(data).map((prob, i) => ({
      gesto: etiquetas[i],
      confianza: (prob * 100).toFixed(2)
    }));


    const resultadoTexto = resultados.map(r =>
      `Gesto ${r.gesto}: ${r.confianza}%`
```

```
    ).join('<br>');

    document.getElementById('resultado').innerHTML = resultadoTexto;
  }
```

Interfaz Web para Análisis de numeros

Esta interfaz fue diseñada para permitir al usuario dibujar un número del 0 al 9 directamente sobre un lienzo (<canvas>) HTML y ejecutar una predicción en tiempo real utilizando TensorFlow.js en el navegador, sin necesidad de subir archivos ni hacer peticiones al backend. El diseño sigue la línea visual unificada del proyecto, con navegación entre secciones, control de acciones y resultados visuales.

Lógica de dibujo en canvas

El usuario dibuja libremente con el mouse sobre un lienzo negro. Se simula el trazo blanco utilizando ctx.strokeStyle = 'white' y se controla con los eventos:

Procesamiento de la imagen y predicción

Cuando el usuario hace clic en "Comprobar", se ejecuta la siguiente lógica:

1. Se convierte el contenido del canvas a una imagen base64.
2. Se crea un objeto Image() y se espera a que cargue.
3. Una vez cargado, se transforma a tensor, redimensiona a 28x28, se convierte a escala de grises y se normaliza.
4. Se realiza la inferencia con modelo.predict(tensor).

Procesamiento de la imagen y predicción

```
async function comprobar() {
    const dataUrl = canvas.toDataURL('image/png');

    const img = new Image();
    img.src = dataUrl;

    img.onload = async () => {
      let tensor = tf.browser.fromPixels(img, 1)
        .resizeNearestNeighbor([28, 28])
        .toFloat()
        .div(255.0)
        .expandDims(0);
```

```
      const prediccion = modelo.predict(tensor);
      const data = await prediccion.data();

      // Construye lista de probabilidades por dígito
      const resultados = Array.from(data).map((prob, i) => ({
        digito: i,
        confianza: (prob * 100).toFixed(2)
      }));

      // Ordenar por confianza descendente (opcional)
      resultados.sort((a, b) => b.confianza - a.confianza);

      // Mostrar todos
      const resultadoTexto = resultados.map(r =>
        `Dígito ${r.digito}: ${r.confianza}%`
      ).join('<br>');

      document.getElementById('resultado').innerHTML = resultadoTexto;
    };
  }
```

Consejos y Buenas Prácticas

A continuación se presentan recomendaciones para garantizar la calidad, escalabilidad y mantenibilidad de proyectos que integran machine learning, servidores web y ejecución en el navegador.

Sobre la creación de datasets

- Establece condiciones constantes de captura, como iluminación, fondo y resolución, para evitar sesgos y mejorar la precisión del modelo.
- Organiza las imágenes en carpetas nombradas por clase (por ejemplo, /0, /1, /feliz, /enojado) para facilitar su uso con bibliotecas como Keras.
- Asegura un número equilibrado de muestras por clase para evitar que el modelo aprenda de forma desbalanceada.

Sobre el entrenamiento del modelo

- Redimensiona y normaliza todas las imágenes antes del entrenamiento.
- Monitorea continuamente las métricas de pérdida (loss) y precisión (accuracy) durante el entrenamiento.
- Guarda el modelo en formatos adecuados según el entorno de ejecución: .h5 para backend, .json para TensorFlow.js, y .keras para compatibilidad futura.

Sobre la implementación del servidor

- Usa Express.js de forma modular para separar rutas, archivos estáticos y vistas.
- Define una estructura clara de carpetas, como /views para las vistas y /public para los archivos accesibles desde el navegador.
- Establece límites en la carga de datos (por ejemplo, 20MB) para prevenir errores al procesar imágenes grandes.

Sobre la interfaz web y renderizado en navegador

- Asegura una experiencia de usuario fluida con botones visibles, mensajes claros y resultados inmediatos.
- Valida que el usuario haya proporcionado una imagen antes de realizar la predicción.
- Carga los modelos de forma asíncrona para evitar bloqueos en la interfaz y mejorar el rendimiento.

Sobre el uso de TensorFlow.js

- Asegúrate de que el modelo esté ubicado en una ruta accesible desde el navegador (por ejemplo, /tfjs_model).
- Normaliza los valores de los píxeles dividiendo entre 255 para mantener consistencia con el entrenamiento.
- Utiliza tf.tidy() cuando sea necesario para evitar saturar la memoria del navegador en sesiones largas.

Recomendaciones generales

- Documenta el código con comentarios que expliquen funciones clave y flujos importantes.
- Utiliza control de versiones (por ejemplo, Git) para llevar registro de cambios y mantener respaldo del proyecto.
- Utiliza archivos de configuración (por ejemplo, config.js o .env) para separar variables como puertos, rutas o credenciales.
- Prueba el sistema en distintos navegadores y dispositivos antes de su despliegue o presentación.